The REST is silence

Restful Webservices mit JAX-RS

Werner Eberling

MATHEMA Software GmbH

Was steckt hinter REST?

HTTP-basiert

leichtgewichtige WebServices

Die Generallösung für lose Kopplung

Das bessere SOAP

Clientside State

Was steckt wirklich hinter REST?

- REpresentational STate Transfer
  - aus: „Architectural styles and the Design Network-based software Architectures“, Roy Fielding, 2000
- Die Architektur des WorldWideWebs
- REST ist ein Architekturstil
  - Soviel zur Frage REST vs. SOAP ;)

Die Grundprinzipien von REST

Einheitliches Interface

Zustandslose Kommunikation

Identifizierbare Ressourcen mit unterschiedlichen Repräsentationen

Verknüpfung / Hypermedia

Selbstbeschreibende Nachrichten

RESTful HTTP

Fahren Sie

auch in den

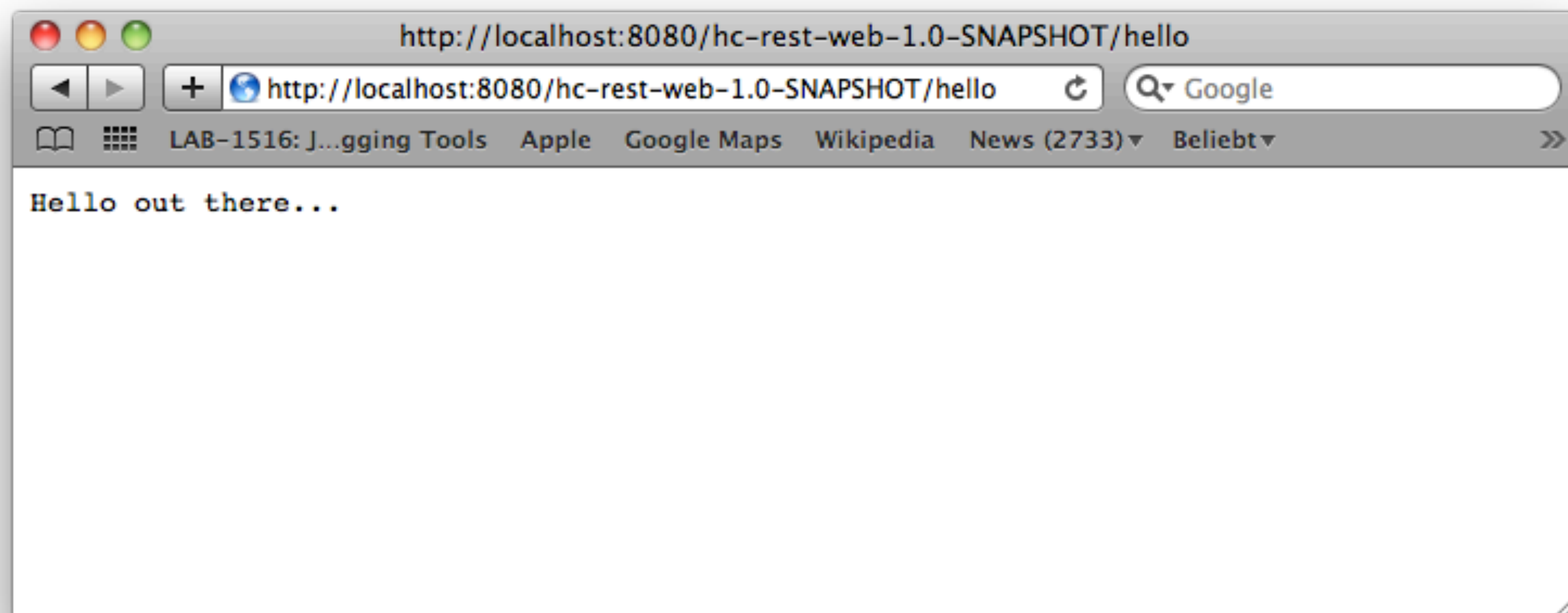
Siebten?

JAX-RS

- Java API for RESTful Web Services (JSR-311)
- Aktuelle Version: 1.1
- Teil der JEE6
  - javax.ws.rs.*
- Implementierungen
  - Jersey (Referenz-Implementierung)
  - JBoss RESTEasy
  - Restlet
  - Apache Wink

Der Klassiker

```
@Path("/hello")
public class HelloResource {

    @GET
    @Produces("text/plain")
    public String sayHello() {
        return "Hello out there... ";
    }

}
```



Zu statisch? Dann also mit Benutzereingabe

```
@Path("/hello")
public class HelloResource {

    @GET
    @Produces("text/plain")
    public String sayHello(@QueryParam("who") String toMe) {
        return "Hello " + toMe + ", how are you out there?";
    }

}
```

Hinter den Kulissen

- Konfiguration der JAX-RS Implementierung
  - hier: Jersey

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
    <servlet>
        <servlet-name>JerseyServlet</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>JerseyServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

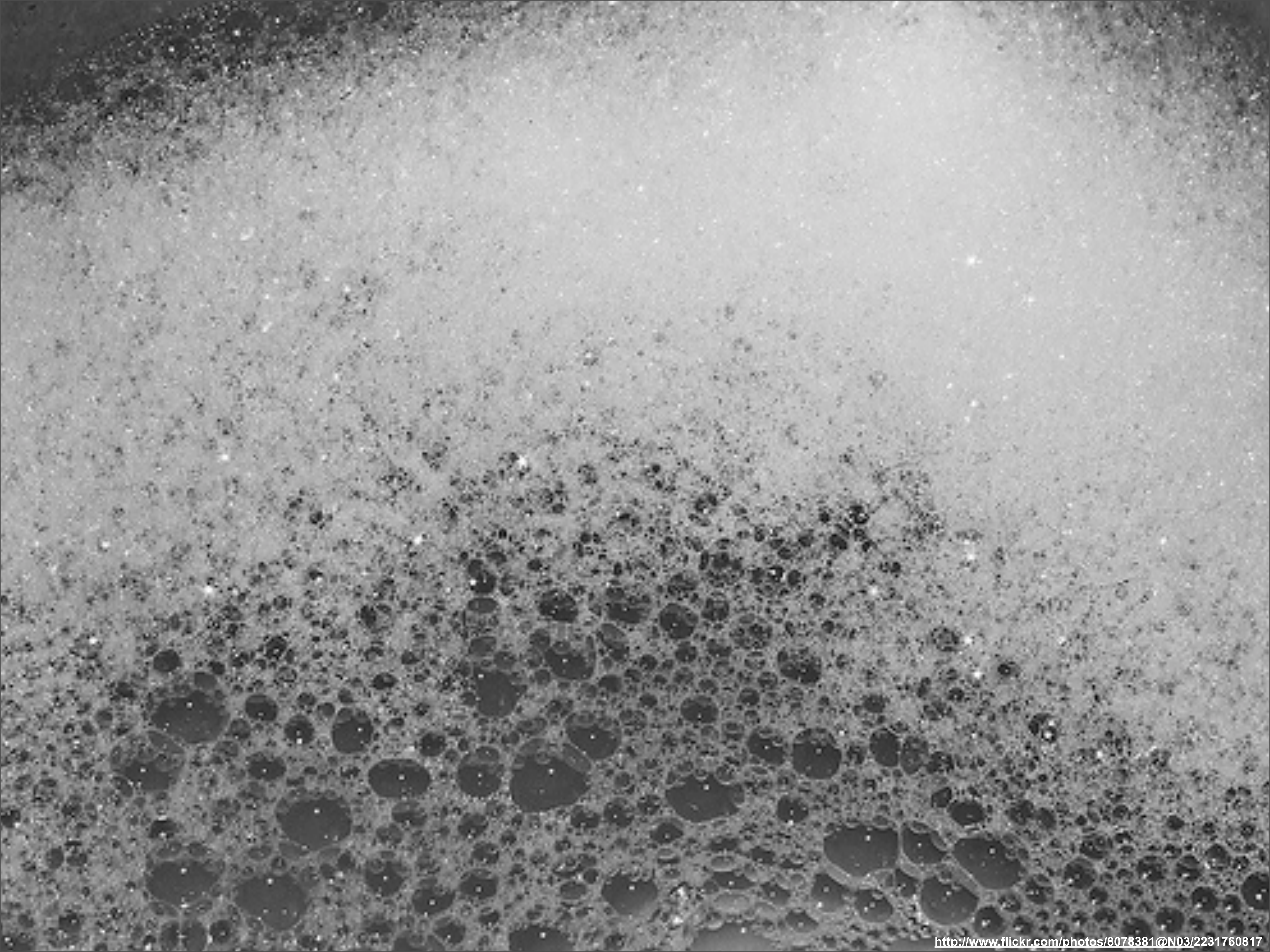
Nächster Schritt: Personalisierte Grüsse

- Anforderungen
  - Erzeugen von Begrüssungen
  - Abrufen von gespeicherten Begrüssungen
  - Löschen von Begrüssungen
  - Auflisten aller gespeicherter Begrüssungen

- Aufbau der Schnittstelle

Erzeugen einer Begrüssung	/createGreeting?name=abcd&greeting=hello
Abrufen einer Begrüssung	/getGreeting?name=abcd
Löschen einer Begrüssung	/deleteGreeting?name=abcd
Abrufen aller Begrüssungen	/getAllGreetings

Vorsicht: HTTP-API aber kein REST!


http://www.flickr.com/photos/8078381@N03/2231760817

SOAPy REST

- Verwendung von JAX-RS zur Implementierung leichtgewichtiger, webbasierter RPCs
- Ermöglicht Clients ohne SOAP-Unterstützung einfachen Webservice-Zugriff
- Grundsätzlich erst einmal nicht's Schlechtes
- Lediglich keine REST-Architektur

Zu tief gestapelt

- Beispiel verwendet HTTP nur für den Transport
  - ISO-OSI-Schicht: 4
  - URIs beschreiben Operationen
- REST verwendet HTTP als Application-Protocol
  - ISO-OSI-Schicht: 7
  - URIs beschreiben Ressourcen
  - Operationen sind Teil des Protokolls
    - PUT - Erzeugung einer Ressource
    - GET - Auslesen einer Ressource
    - POST - Verändern einer Ressource
    - DELETE - Löschen einer Ressource

Noch einmal personalisierte Grüsse

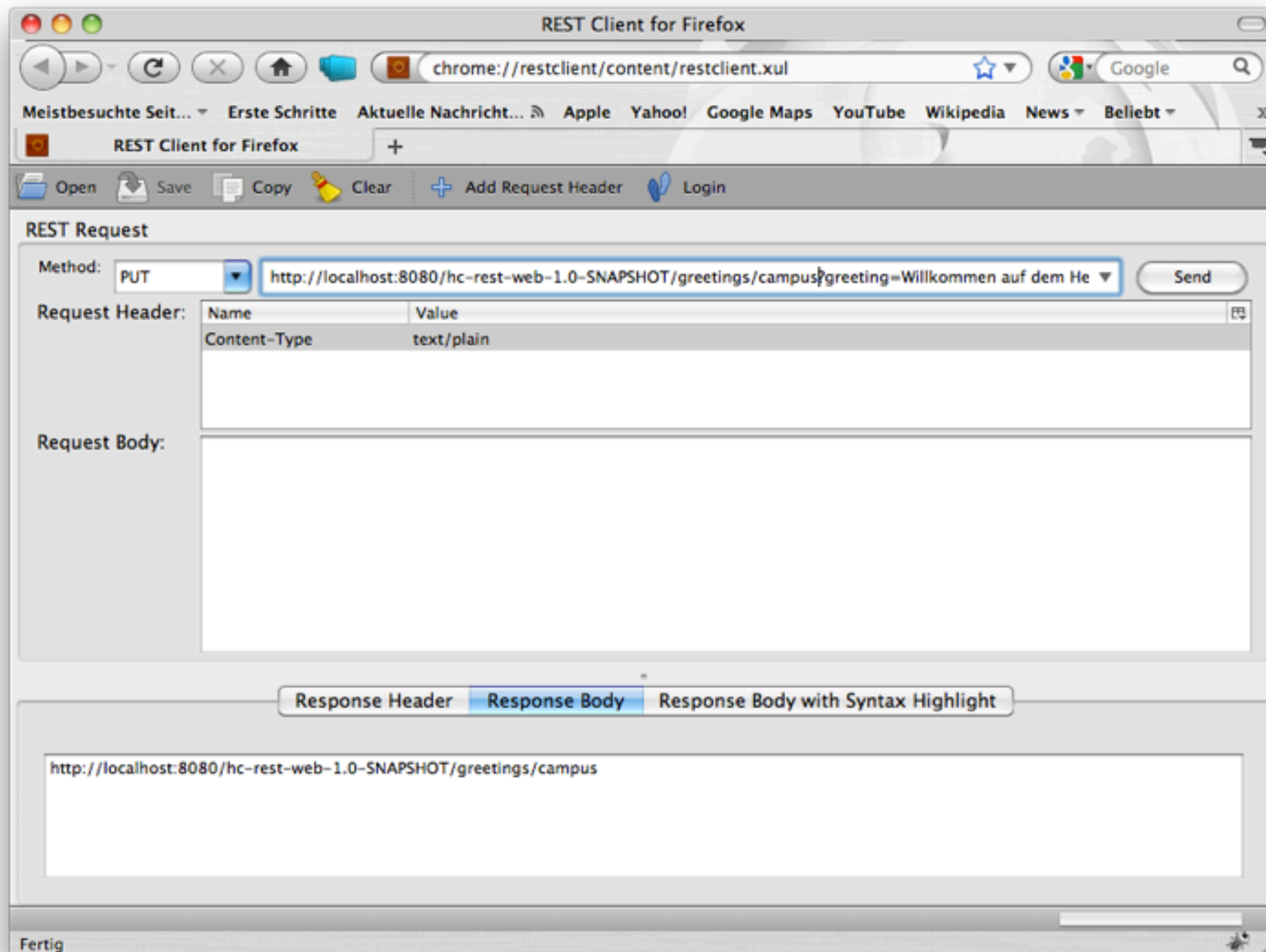
- Name der Ressource: Greeting
- Operationen
  - Erzeugen
  - Abrufen
  - Löschen
- Aufbau der Schnittstelle

Erzeugen einer Begrüssung	PUT /greetings/abcd?greeting=hello
Abrufen einer Begrüssung	GET /greetings/abcd
Löschen einer Begrüssung	DELETE /greetings/abcd
Abrufen aller Begrüssungen	GET /greetings

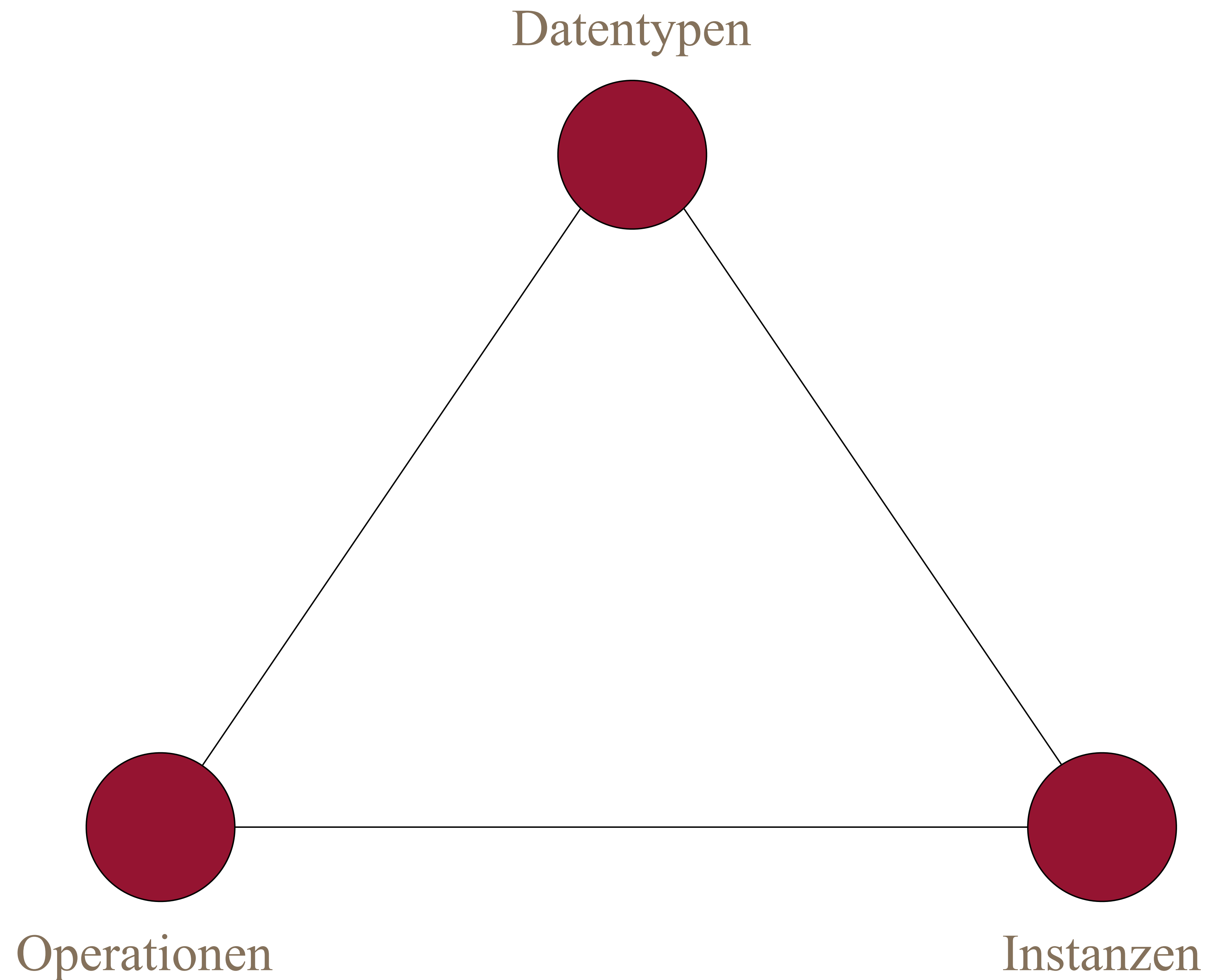
So sieht‘s im Code aus

```
@Path("/greetings")
public class GreetingResource {
    @Context UriInfo uinfo;

    @PUT
    @Path("/{greetingId}")
    @Produces("text/uri-list")
    public String addNewGreeting(@PathParam("greetingId") String greetingId,
                                 @QueryParam("greeting") String greeting) {
        storedGreetings.put(greetingId, greeting);
        return uinfo.getBaseUriBuilder().path(getClass()).path(getClass(),"greet").
                    build(greetingId).toASCIIString();
    }

    @GET
    @Path("/{greetingId}")
    @Produces("text/plain")
    public String greet(@PathParam("greetingId") String greetingId) {
        return storedGreetings.get(greetingId);
    }

    @GET
    @Produces("text/uri-list")
    public String allGreetings() {
        return storedGreetings.toUriList();
    }
}
```

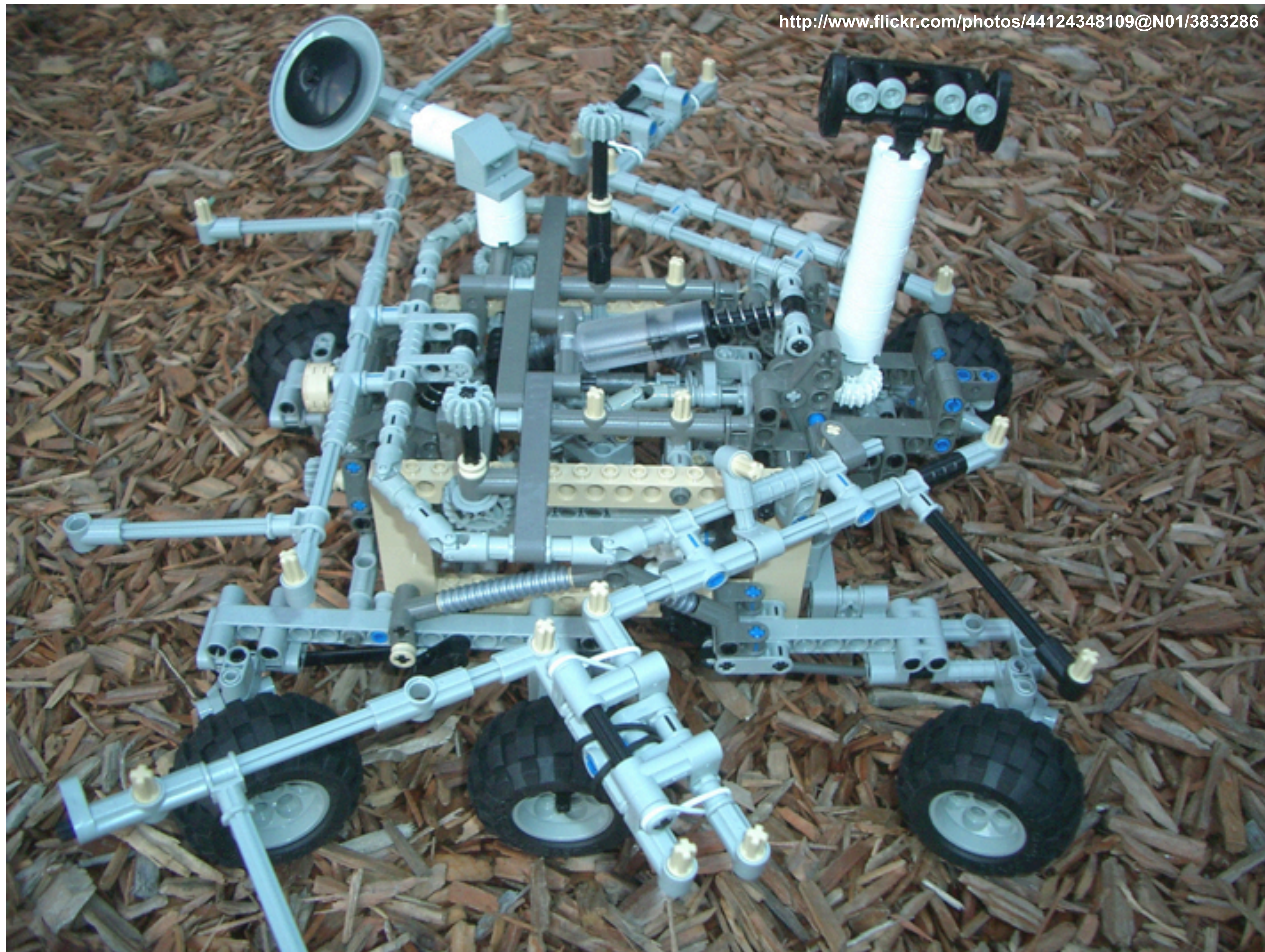
REST-Client als Firefox-Plugin



Options Dreieck



Und wenn's komplexer wird?



Der nächste Klassiker: Kundenverwaltung

- Name der Ressource: Customer
- Operationen
  - Erzeugen
  - Abrufen
  - Ändern
- Aufbau der Schnittstelle

Erzeugen eines Kunden	PUT /customers?name=Doe&firstname=John
Abrufen eines Kunden	GET /customers/4711
Ändern eines Kunden	POST /customers/4711 (Daten im Body)
Abrufen aller Kunden	GET /customers

Der Kunde als komplexer Datentyp (1/2)

- Serverseitig: Persistente Klasse
  - JPA-Entity
- Übertragung als XML
  - Content-Type: application/xml
- Mapping über JAX-B
  - @XMLRootElement
  - JavaBeans-Konventionen

Der Kunde als komplexer Datentyp (2/2)

```
@XmlRootElement
@Entity
public class Customer implements Serializable {
    @Id @GeneratedValue
    private long customerNumber;
    private String name;
    private String firstname;
    @Transient
    private String link;

    public Customer() {
    }

    public long getCustomerNumber() {
        return customerNumber;
    }

    public void setCustomerNumber(long customerNumber) {
        this.customerNumber = customerNumber;
    }

   // weiter Konstruktoren und Methoden...
}
```

Die Ressource als Stateless EJB (1/2)

```
@Stateless
@Remote
public class CustomerResourceImpl implements CustomerResource {

    @PersistenceContext EntityManager entityManager;

    @Context UriInfo uinfo;

    public Customer createCustomer(String name, String firstname) {
        return createCustomer(new Customer(name, firstname));
    }

    public Customer createCustomer(Customer customer) {
        entityManager.persist(customer);
        UriBuilder builder = getUriBuilder().path(
                CustomerResource.class, "findCustomerByNumber");
        customer.setLink(builder.build(customer.getCustomerNumber())
                .toASCIIString());
        return customer;
    }

    public Customer findCustomerByNumber(long customerNumber) {
        return entityManager.find(Customer.class, customerNumber);
    }

   // weiter Methoden...
}
```

Die Ressource als Stateless EJB (2/2)

- Nutzt die Vorteile des EJB-Containers
- Aber wo sind die JAX-RS-Annotationen?

```
@Path("/customers")
public interface CustomerResource {

    @PUT @Consumes("text/plain") @Produces("application/xml")
    public Customer createCustomer(@QueryParam("name") String name,
                                   @QueryParam("firstname") String firstname);

    @PUT @Consumes("text/xml,application/xml") @Produces("application/xml")
    public Customer createCustomer(Customer customer);

    @GET @Path("/{id}") @Produces("application/xml")
    public Customer findCustomerByNumber(@PathParam("id") long customerNumber);

    @GET @Produces("application/xml")
    public List<Customer> getAll();

    @POST @Consumes("text/xml,application/xml") @Produces("application/xml")
    public Customer changeCustomer(Customer customer);
}
```

Nochmal kurz hinter den Kulissen

- Konfiguration der JAX-RS Implementierung
  - hier: Jersey

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
    <servlet>
        <servlet-name>JerseyServlet</servlet-name>
        <servlet-class>
            com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <init-param>
        <param-name>com.sun.jersey.config.property.packages</param-name>
        <param-value>de.mathema.intern.rest;de.mathema.intern.rest.ejb</param-value>
    </init-param>
    <servlet-mapping>
        <servlet-name>JerseyServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Wenn's etwas mehr sein soll...

- javax.ws.rs.core.Context
- javax.ws.rs.core.HttpHeaders
- javax.ws.rs.core.Request
- javax.ws.rs.core.Response
- javax.ws.rs.core.SecurityContext
- javax.ws.rs.core.UriInfo

Und was ist mit dem Client?

- JAX-RS adressiert die Server-Seite
- Client-Seite wird vernachlässigt
- JAX-RS-Implementierungen helfen hier aus
  - Jersey
  - RESTEasy
  - ...

```
...
  Client client = Client.create();
  WebResource webResource =
                client.resource("http://localhost:8080/hc-rest-web-1.0-SNAPSHOT/customers");
  Customer c = webResource.get(Customer.class);
...
```

Fazit

- JAX-RS ermöglicht einfach Implementierung von Webservices
  - Lediglich Servlet-Konfiguration erscheint unnötig
  - Mit dem richtigen Architekturansatz auch RESTful ,)
- REST-Ansatz bietet eine interessante Alternative zu klassischen RPC-Architekturen
- ABER: Es hängt von den Anforderungen ab
  - REST ist nicht die Lösung aller Probleme!

Vielen Dank!

Werner Eberling
MATHEMA Software GmbH